CS111

Rishab Ketan Doshi

Overview

Logistics

File Manipulation

GetOPT

Makefiles

Exception Handling

GDB

Project Details

Prof. Xu handles coursework, All TA's responsible for projects **only**.

10 Projects to be submitted - every week a new project is released

Note the order of projects -

P0, P1A, P1B, **P4A**, P2A, P2B, P4B, P3A, P3B, P4C

Project 4 requires a hardware device - Beaglebone.

P4A is early on in the quarter, please purchase immediately, details on website.

Project Details (contd.)

Please redownload your submission and verify that your submitted file meets the specs

This is very important, have seen multiple cases where students submit empty tar files, or incorrectly named files.

The automation script that grades your project will fail and a **ZERO** will be awarded in this case.

Lateness Policy

- Let N be the number of days late (specifically the ceiling of number of hours late divided by 24). Then the student is deducted 2^{N-1} percent off of their grade.
- Even 30s late is considered 1 day late.

Submit within 1 day after deadline	1% deducted
Between 1 and 2 days of deadline	2% deducted
Between 2 and 3 days of deadline	4% deducted
Between 3 and 4 days of deadline	8% deducted
Between 4 and 5 days of deadline	16% deducted
Between 5 and 6 days of deadline	32% deducted
Between 6 and 7 days of deadline	64% deducted
Beyond 7 days	No credit

TA <--> Projects

TA's will be working in pairs on projects to help clarify queries and grade your submissions. For regrade requests, please reach out to respective TA's

Howard and Shaan: Lab 0, 1A, 1B and 4A

- howardx@cs.ucla.edu
- shaankaranmathur@gmail.com

Rishab and Nikita: Lab 2A, 2B and 4C

- nikita3096@g.ucla.edu
- rishabkdoshi@g.ucla.edu

Karen and Alex: Lab 3A, 3B and 4B

- alexandre.tiard@gmail.com
- quadroskaren369@gmail.com

Project Queries

All queries regarding project implementation to be posted on the class Piazza group.

Before you post a query:

- Please study the Project Spec, it is extremely likely your question has already been answered.
- Please read through other piazza posts in the project folder, some student might have already asked a similar question.

Please participate and answer questions, there are 270 students and only 2 TA's(per project) monitoring piazza.

Office Hours

Where?

Boelter 3256S

When?

Only next week: Tuesday - 08.30 - 10.30 AM

From week 3 onwards:

office hours: Monday - 09.30 - 11.30 AM

Development Environment Setup

*"All grading will **only** be performed on lnxsrv09.seas.ucla.edu unless otherwise stated. Make sure you only test your code on that server. Please keep in mind that a submission compiles and runs on Mac/Ubuntu/Windows does not mean it works on lnxsrv09!"*

Everyone should have a SEAS account and access to lnxsrv09

Logging in:

```
ssh username@lnxsrv09.seas.ucla.edu
```

Please ensure your PATH is prepended with **/usr/local/cs/bin**, there are multiple variants of different libraries(gcc, libc etc) on all servers, this will ensure consistency with GRADING scripts.

How to verify? Piazza post

Check output of **which** commands →

```
[rishabd@lnxsrv09 ~]$ which gdb
/usr/local/cs/bin/gdb
[rishabd@lnxsrv09 ~]$ which make
/usr/local/cs/bin/make
[rishabd@lnxsrv09 ~]$ which gcc
/usr/local/cs/bin/gcc
```

Project 0

Project 0 - Why?

Serves as a test for **you** to evaluate if you are equipped to take this course.

If you find this project difficult, consider taking this course at a later time after gaining mastery over concepts in CS32, CS33 and CS35L.

Project 0 - Key Objectives

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard Makefile.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

Project 0

write a program that copies its standard input to its standard output by *read(2)*-ing from file descriptor 0 (until encountering an end of file) and *write(2)*-ing to file descriptor 1. If no errors (other than EOF) are encountered, your program should *exit(2)* with a return code of 0.

Your program executable should be called lab0, and accept the following (optional) command line arguments (in any combination or order):

- **--input=***filename* ... use the specified file as standard input (making it the new fd0).
  If you are unable to open the specified input file, report the failure (on stderr, file descriptor 2) using *fprintf(3)*, and *exit(2)* with a return code of 2.
- **--output=***filename* ... create the specified file and use it as standard output (making it the new fd1).
  If you are unable to create the specified output file, report the failure (on stderr, file descriptor 2) using *fprintf(3)*, and *exit(2)* with a return code of 3.
- **--segfault** ... force a segmentation fault (e.g., by calling a subroutine that sets a char * pointer to NULL and then stores through the null pointer). If this argument is specified, do it immediately, and do not copy from stdin to stdout.
- **--catch** ... use *signal(2)* to register a SIGSEGV handler that catches the segmentation fault, logs an error message (on stderr, file descriptor 2) and *exit(2)* with a return code of 4.

Files

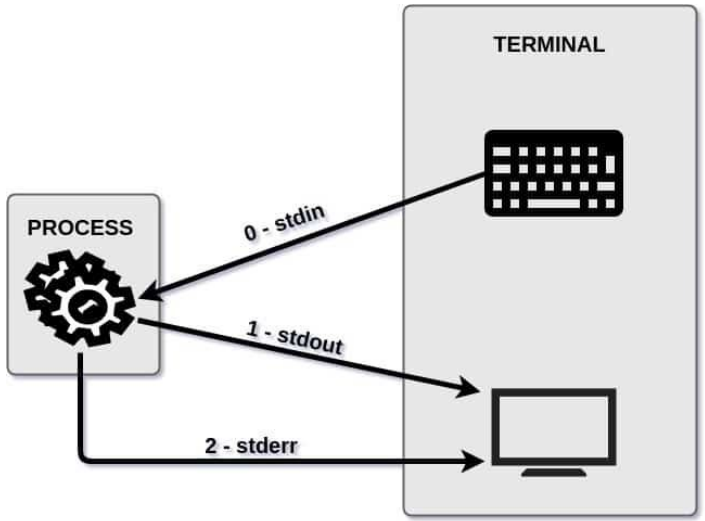
Everything is a file

In UNIX everything is a file.

- Keyboard is a file that is read only for the kernel
- Screen is a file that is write only for the kernel
- Sockets are special files that are used for sending and receiving data

Whenever a *PROCESS* **opens** a file, the kernel allocates a number that is used to “refer” to that file in subsequent **read, write** operations.

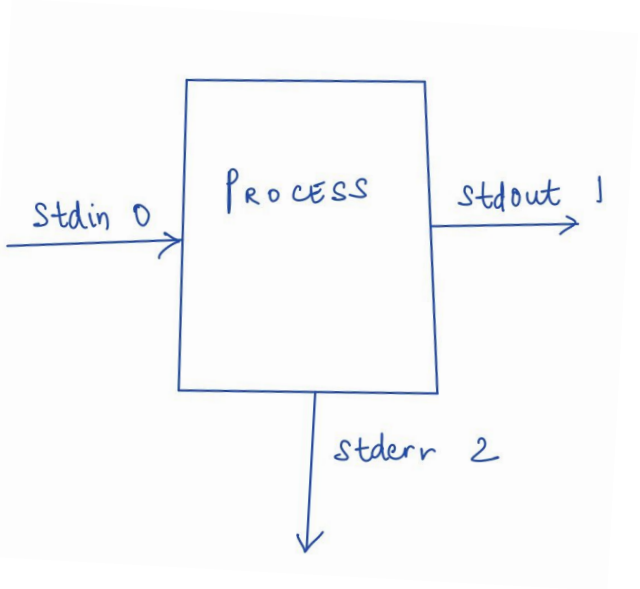
This number is called as the File descriptor



File Descriptors

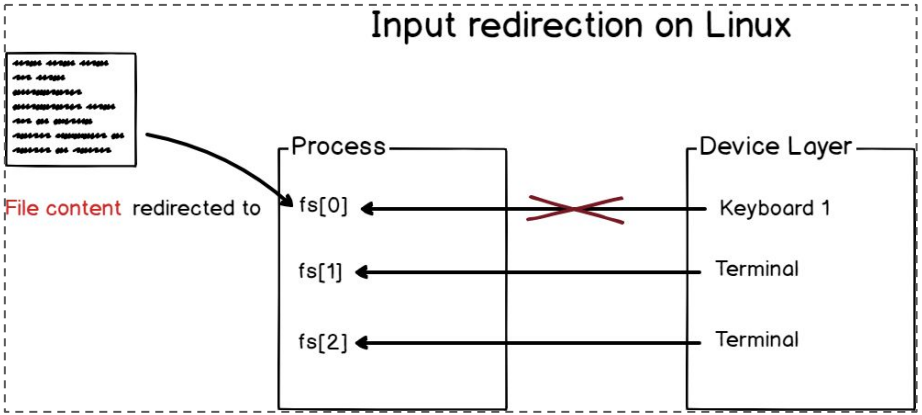
For any PROCESS (program in execution)

- File Descriptors (FD) are **non-negative integers** (0, 1, 2, ...) that are associated with files that are opened by the process.
- There are certain standard FD's that are **opened by default** when a program starts, they are:
  - STDIN_FILENO - 0
  - STDOUT_FILENO - 1
  - STDERR_FILENO - 2
- For any newly opened files, FD's are allocated in the **sequential** order, meaning the lowest possible unallocated integer value.

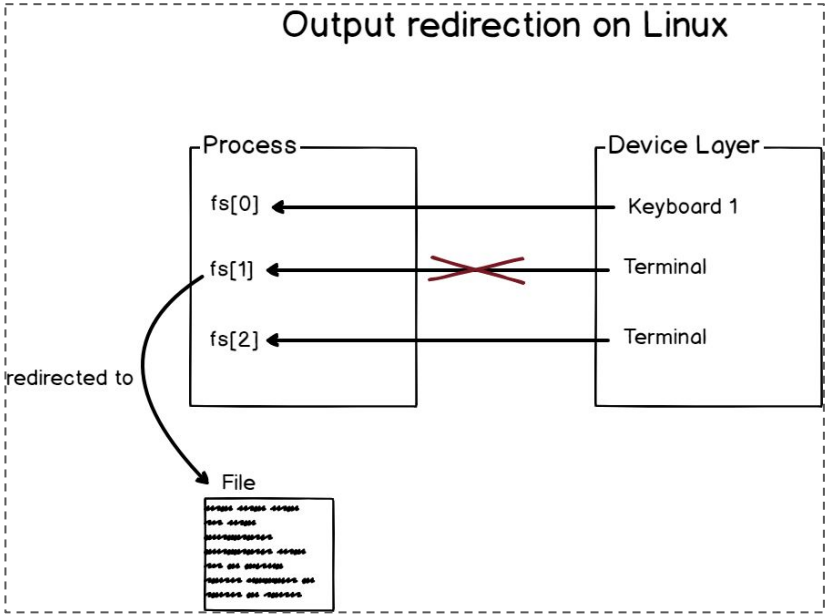


Input Output Redirection - Command Line

`sort < domains`



`echo devconnected > file`



POSIX file operations

A set of operations for manipulating files on POSIX compliant systems.

These operations should work consistently across various implementations of UNIX.

In this project we will use the below operations

open(2), creat(2), close(2), dup(2), read(2), write(2)

Detour - man pages

What are the numbers on the commands that we have been seeing?

i.e., what is 2 in open(2), creat(2), dup(2), close(2), read(2), write(2)

The number corresponds to what section of the manual that page is from.

Detour - man pages

- The UNIX Manual has multiple sections, running **`man man`** outputs the below message.

```
The table below shows the section numbers of the manual followed by the types of pages they contain.

       1   Executable programs or shell commands
       2   System calls (functions provided by the kernel)
       3   Library calls (functions within program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions eg /etc/passwd
       6   Games
       7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
       8   System administration commands (usually only for root)
       9   Kernel routines [Non standard]

       A manual page consists of several sections.
```

- To open a man page from a specific section ***n***, use the command : man ***n*** command

  `man 1 printf` to read documentation for print(1)

  `man 3 printf` to read documentation for print(3)

Back to POSIX File Operations

open(2) - opens a file at specified path and mode(read, write, read & write)

creat(2) - creates a file, but can’t open an existing file (will overwrite if creat is called on existing path)

close(2) - closes a file, takes the file descriptor as argument to the function

read(2) - reads specified no. of bytes from the current file offset to the buffer

write(2) - writes specified no. of bytes from the buffer, moving the file offset by those many bytes

Read about all of these functions in more detail by going through the man page and understanding the exact arguments and return values.

dup(2)

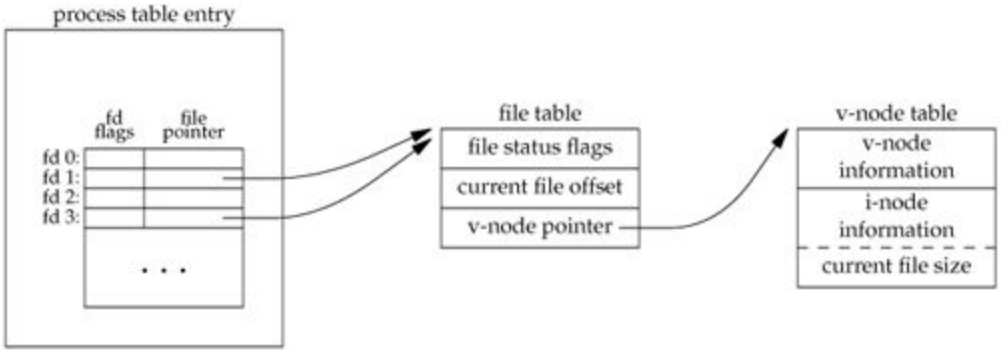
```
int dup(int oldfd);
```

These system calls create a copy of the file descriptor oldfd.

dup() uses the lowest-numbered unused descriptor for the new descriptor.

After a successful return from dup, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see open(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the descriptors, the offset is also changed for the other.

dup(2)



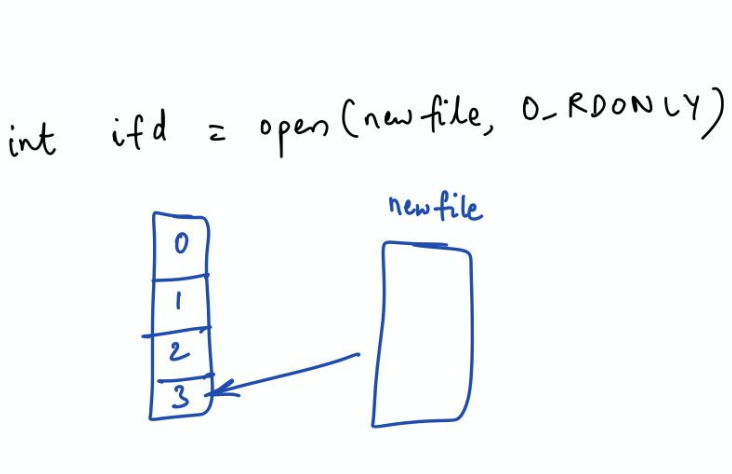

```
newfd = dup(1);
```

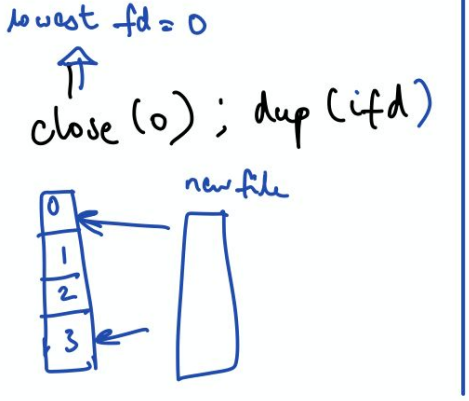
Here we are duplicating the file descriptor 1(stdin).

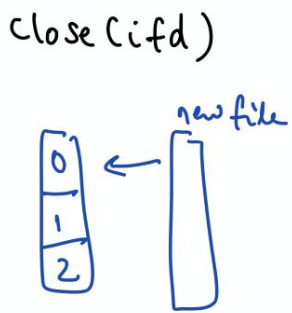
Remember that dup assigns the new file descriptor to the next available lowest file descriptor, which is 3, since 0,1,2 are opened by default for every process.

Input Redirection - C Program

```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

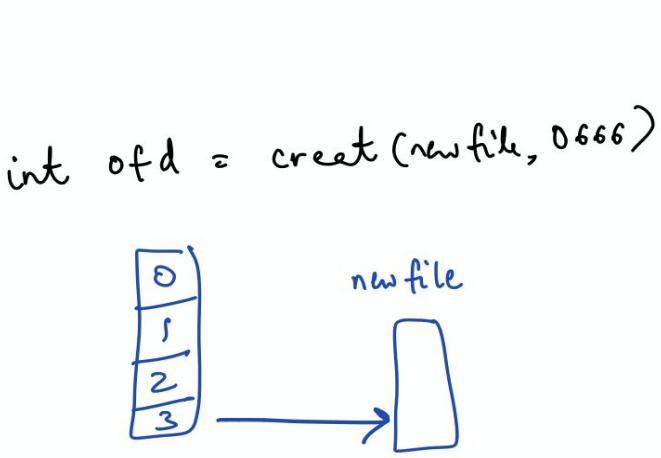


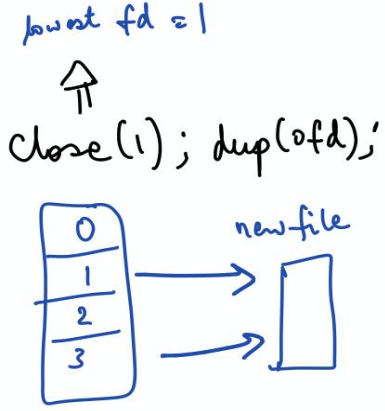


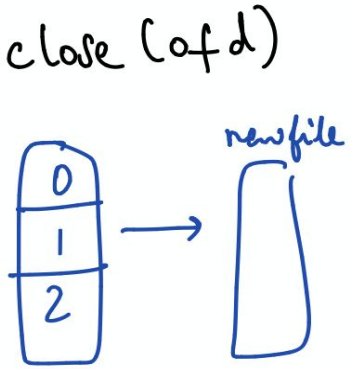


Output Redirection - C Program

```
int ofd = creat(newfile, 0666);
if (ofd >= 0) {
        close(1);
        dup(ofd);
        close(ofd);
}
```







Project 0

write a program that copies its standard input to its standard output by *read(2)*-ing from file descriptor 0 (until encountering an end of file) and *write(2)*-ing to file descriptor 1.

If no errors (other than EOF) are encountered, your program should exit(2) with a return code of 0. → refer to man page of exit(2)

Your program executable should be called lab0, and accept the following (optional) command line arguments (in any combination or order):

- **--input=***filename* ... use the specified file as standard input (making it the new fd0).
  If you are unable to open the specified input file, report the failure (on stderr, file descriptor 2) using *fprintf(3)*, and *exit(2)* with a return code of 2.
- **--output=***filename* ... create the specified file and use it as standard output (making it the new fd1).
  If you are unable to create the specified output file, report the failure (on stderr, file descriptor 2) using *fprintf(3)*, and *exit(2)* with a return code of 3.
- **--segfault** ... force a segmentation fault (e.g., by calling a subroutine that sets a char * pointer to NULL and then stores through the null pointer). If this argument is specified, do it immediately, and do not copy from stdin to stdout.
- **--catch** ... use *signal(2)* to register a SIGSEGV handler that catches the segmentation fault, logs an error message (on stderr, file descriptor 2) and *exit(2)* with a return code of 4.

Project 0 - Key Objectives

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard Makefile.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

Command Line Argument Parsing

Detour - Extern Variables

variables that are external to all functions, that is, variables that can be accessed by name by any function.

Keypoint:

external variables are globally accessible, they can be used instead of argument lists to communicate data between functions

In our use-case, we will use many external variables that are set by different processes.

Example:

- errno(3) is an extern variable set to represent common error numbers. You will use errno(3) in combination with strerror(3) to print informative exit messages. Simple example - errno.h
- We will also be using other extern variables like optind, optarg, opterr for command line parsing using getopt(3)
- Additional explanation https://github.com/rishabkdoshi/CS111/blob/master/week1/extern.md

getopt

getopt is an API provided to parse command line arguments.

It is feature rich supporting long and short options, required, optional and no arguments

It will take time to understand the API, will require multiple readings of the man page

Demystifying getopt

```
int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;

extern int optind, opterr, optopt;
```

Demystifying getopt

The getopt() function parses the command-line arguments. Its arguments argc and argv are the argument count and array as passed to the main() function on program invocation. An element of argv that starts with '-' (and is not exactly "-" or "--") is an option element. The characters of this element (aside from the initial '-') are option characters. If getopt() is called repeatedly, it returns successively each of the option characters from each of the option elements.

The variable optind is the index of the next element to be processed in argv. The system initializes this value to 1. The caller can reset it to 1 to restart scanning of the same argv, or when scanning a new argument vector.

If getopt() finds another option character, it returns that character, updating the external variable optind and a static variable nextchar so that the next call to getopt() can resume the scan with the following option character or argv-element.

optstring is a string containing the legitimate option characters. If such a character is followed by a colon, the option requires an argument, so getopt() places a pointer to the following text in the same argv-element, or the text of the following argv-element, in optarg.

By default, getopt() permutes the contents of argv as it scans, so that eventually all the nonoptions are at the end.

Run

Now we are going to run the code in the man page example.

Here is `getopt_example1.c`, same program with more logging - so you understand what is going on.

Run Commands

Compile the program with

```
gcc -o example1 getopt_example1.c
```

Run the program with different arguments and try to understand what is going on (~2 mins)

- `./example1 -t 2 -n rkd`
- `./example1 -n rkd -t 2`
- `./example1 -t`
- `./example1 -n`
- `./example1 -n -t 2`
- `./example1 -name rkd -t 2`

Demystifying getopt

```
The  getopt()  function parses the command-line arguments.  Its arguments argc and argv are the
argument count and array as passed to the main() function on program invocation
```

Refer to the output of the function `printArgs` and verify that the above statements are true

Demystifying getopt

```
An element of argv that starts with '-' (and is not exactly "-" or "--") is an option element.  The
characters of this element (aside from the initial '-') are option characters.
```

Command	Option Elements	Option Characters
`./example1 -t 2 -n rkd`	t, n	t, n
`./example1 -n rkd -t 2`	t, n	t, n
`./example1 -t`	t, n	t, n
`./example1 -n`	t, n	t, n
`./example1 -n -t 2`	t, n	t, n
`./example1 -name rkd -t 2`	t, n	t, n ,a, m, e

Demystifying getopt

```
If  getopt()  is  called repeatedly, it returns successively each of the option characters from each of the option elements.
```

Uncomment the below line from getopt_example1.c

```
//printAllOptionCharacters(argc,argv); exit(1);
```

Compile and run the program with **`./example1 -name rkd -t 2`**

OUTPUT

```
[INFO] printing all option characters
[INFO] iteration=0 opt=n optarg=(null) optind=1
./example1: invalid option -- 'a'
[INFO] iteration=1 opt=? optarg=(null) optind=1
./example1: invalid option -- 'm'
[INFO] iteration=2 opt=? optarg=(null) optind=1
./example1: invalid option -- 'e'
[INFO] iteration=3 opt=? optarg=(null) optind=2
[INFO] iteration=4 opt=t optarg=2 optind=5
[INFO] Done
```

Demystifying getopt

```
The  variable  optind is the index of the next element to be processed in argv.  The system
initializes this value to 1.  The caller can reset it to 1 to restart scanning of the same argv, or
when scanning a new argument vector.
```

Output of ./example1 -t 2 -n rkd

```
[INFO] State of extern variables before calling getopt
[INFO] iteration=-1 opt=? optarg=(null) optind=1
...
[INFO] iteration=0 opt=t optarg=2 optind=3
[INFO] iteration=1 opt=n optarg=(null) optind=4
```

Why is it initialized to 1?

The first element in argv, i.e., index 0 is always going to be the program to be executed. i.e., in this case argv[0] is going to be example1

Demystifying getopt

If getopt() finds another option character, it returns that character, updating the external variable optind and a static variable nextchar so that the next call to getopt() can resume the scan with the following option character or argv-element.

Output of ./example1 -t 2 -n rkd

```
[INFO] State of extern variables before calling getopt
[INFO] iteration=-1 opt=? optarg=(null) optind=1
...
[INFO] iteration=0 opt=t optarg=2 optind=3
[INFO] iteration=1 opt=n optarg=(null) optind=4
```

Demystifying getopt

```
optstring  is a string containing the legitimate option characters.  If such a character is
followed by a colon, the option requires an argument, ...
```

In our case optstring was nt:

```
(opt = getopt(argc, argv, "nt:")
```

So, t requires an argument.

```
..., so getopt() places a pointer to the following text in the same argv-element, or the text of the
following argv-element, in optarg.
```

both ways of setting t's values are valid

-t 24 and -t24

Demystifying getopt

```
By  default,  getopt()  permutes the contents of argv as it scans, so that eventually all
the nonoptions are at the end.
```

Output of ./example1 -t 2 -n rkd

```
...
[INFO] iteration=0 opt=t optarg=2 optind=3
[INFO] iteration=1 opt=n optarg=(null) optind=4
```

Why is optind=3 after 1st example and optind=2 after 2nd example?

Output of ./example1 -n rkd -t 2

```
...
[INFO] iteration=0 opt=n optarg = (null) optind=2
[INFO] iteration=1 opt=t optarg = 2 optind=5
```

Because nonoptions are moved to the end, so for 2nd example

-n rkd -t 2 is changed to -n -t 2 rkd

getoptlong

The getopt_long() function works like getopt() except that it also accepts long options, started with two dashes. (If the program accepts only long options, then optstring should be specified as an empty string (""), not NULL.) Long option names may be abbreviated if the abbreviation is unique or is an exact match for some defined option. A long option may take a parameter, of the form --arg=param or --arg param.

getoptlong specifying options

longopts is a pointer to the first element of an array of struct option declared in <getopt.h> **(show the struct from man)**

The meanings of the different fields are:

name is the name of the long option.

has_arg is: no_argument (or 0) if the option does not take an argument; required_argument (or 1) if the option requires an argument; or optional_argument (or 2) if the option takes an optional argument.

flag specifies how results are returned for a long option. If flag is NULL, then getopt_long() returns val. (For example, the calling program may set val to the equivalent short option character.) Otherwise, getopt_long() returns 0, and flag points to a variable which is set to val if the option is found, but left unchanged if the option is not found.

val is the value to return, or to load into the variable pointed to by flag.

The last element of the array has to be filled with zeros.

Demystifying getoptlong

Download this example and compile the program as below

```
gcc -o example2 getopt_example2.c
```

Try out the below commands:

```
./example2 --uid 305 --name rkd

./example2 --uid 305 --name rkd --senior

./example2 --junior
```

Demystifying getoptlong options

```
struct option long_options[] = {

    {"uid", required_argument, NULL, 'u'}, //what is the students UID?

    {"name", required_argument, 0, 'n'}, //what is the students name?

    {"senior", no_argument, &year_flag, SENIOR}, //is the student a senior

    {"junior", no_argument, &year_flag, JUNIOR}, //is the student a junior

    {NULL, 0, NULL, 0}//end of long options

};
```

Demystifying getoptlong options

```
longopts is a pointer to the first element of an array of struct option declared in <getopt.h>

The meanings of the different fields are:

       name is the name of the long option.

       has_arg is: no_argument (or 0) if the option does not take an argument; required_argument
(or 1) if the option requires an argument; or optional_argument (or 2) if the option takes an
optional argument.

     flag specifies  how  results  are returned for a long option.  If flag is NULL, then
getopt_long() returns val.  (For example, the calling program may set val to the equivalent short
option character.)  Otherwise, getopt_long() returns 0, and flag points to a variable which is set
to val if the option is found, but left unchanged if the option is not found.

     val is the value to return, or to load into the variable pointed to by flag.

The last element of the array has to be filled with zeros.
```

Command Line Argument - Summary

Loop over all command line arguments using getoptlong

Store them in appropriate variables, so that you know how your program should behave.

Go through these examples again, read the man page again, follow up with me during office hours if you have more questions

Project 0 - Key Objectives

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard Makefile.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

Exception Handling - (Signal handling in C)

Exception Handling - C++ (Recap from CS31)

Note that this slide is just for demonstration purposes, we don't use try catch in our projects.

Lets look at this simple example from C++

```
try {
     //some condition fails - throw an exception
     if(b==0) throw DivideByZero();
     c = a / b;
} catch (DivideByZero e) {
     //log error message - b can't be zero
}
```

Exception Handling - C

Using the command signal(2) we can register a signal handler that is to be invoked when the program receives the specified signal.

Example from wikipedia →

```
static void catch_function(int signo) {
    puts("Interactive attention signal caught.");
}

int main(void) {
    if (signal(SIGINT, catch_function) == SIG_ERR) {
        fputs("An error occurred while setting a signal handler.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Raising the interactive attention signal.");
    if (raise(SIGINT) != 0) {
        fputs("Error raising the signal.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Exiting.");
    return EXIT_SUCCESS;
    // exiting after raising signal
}
```

Exception Handling - an analogy

In C++; we use **catch** block to handle an exception

In C; we use **signal(2)** to register a handler function to handle a signal.

Re-emphasizing

In C++; our catch block can handle **expected Exceptions** like DivideByZero, if we have *“registered”* a catch block for that Exception.

In C; our program can handle **expected signals**, if we have registered a handler for that signal using the signal(2) command, that handler will be invoked.

Project 0 - Key Objectives

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard Makefile.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

strerror(3) and exit(2)

Use `strerror(3)` along with the `extern variable errno` to print informative messages.

Use exit(2) to exit with a specified status

Actual usage of strerror and exit left as an exercise.

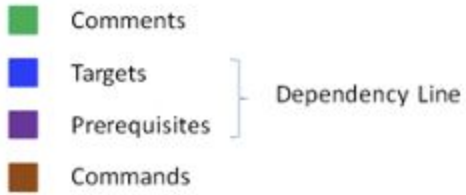
Makefile

- Utility for managing software projects
- Typical use-cases:
  - Compiling code binaries by including requisite dependencies
  - Avoid re-compiling/re-building unchanged files, especially important in a large project where there are multiple dependencies and we want to rebuild only changed dependencies
- Other use-cases:
  - Define sequence of commands(recipe) that have to be executed for a specific command

Good resource on Makefiles - https://www.gnu.org/software/make/manual/html_node/index.html#SEC_Contents

Makefile Example (from CS35L)

```
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
        g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
        g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
        g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
        g++ -g -Wall -c shop.cpp
clean :
        rm -f item.o shoppingList.o shop.o shop
```



Makefile - Key points

1) The command to be run for a target can be **any valid unix command(s).**

For example, if I want to list all files in the directory when I type make list, my makefile will look like

```
list:
	ls
```

Notice that I said valid unix command**s**, if I want to echo info messages before and after running the list command, my makefile will be

```
list:
	echo “Files in current directory are:”
	ls
	echo “Done listing files in current directory”
```

Makefile - Key points

2) By default, make starts with the first target (not targets whose names start with '.'). This is called the default goal. You can execute the default goal by just typing Make on the command line.

```
list:
	ls
listAll:
	ls -a
```

For the above file the default goal is **list**, whereas for the below file, the default goal is **listAll**

```
.list:
	ls
listAll:
	ls -a
```

Smoke Tests

To verify that your program behaved as expected you are to write smoke tests.

These smoke tests will be run using the make check command.

In your Makefile create a goal called as check, that will run these smoke tests.

Hint for smoke tests:

How do you verify the exit status of a shell command?

GDB

Please refer to these slides from CS35L for a detailed explanation of GDB

Project 0 - Key Objectives

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard Makefile.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.